**Data Structures**

**(CSE22101)**

**Assignment 6:**

**Saken Kenzhegulov**

**(20132027)**

**HashMap**

**// constructor: creating a hash table with capacity c, and initializing it**

template <class HashMapElemType>

HashMap<HashMapElemType>::HashMap(unsigned int c) {

capacity = c;

ht = new HashMapElemType\*[capacity];

for(int i=0; i<capacity; i++) ht[i] = NULL; }

**// destructor: deleting all the pointers that we used in hash table and hash table itself**

template <class HashMapElemType>

HashMap<HashMapElemType>::~HashMap() {

for(int i=0; i<capacity; i++)

if(ht[i] != NULL) {

HashMapElemType \*prev = NULL;

HashMapElemType \*curr = ht[i];

while(curr != NULL) {

prev = curr;

curr = curr->link;

delete prev; }

}

delete [] ht; }

**// function for searching our key in hash table**

template <class HashMapElemType> HashMapElemType\*

HashMap<HashMapElemType>::find(const KeyType k) {

unsigned int v = hashfunction(k); **// calculating hash value**

unsigned int i = v%capacity; **//’i’ will store our needed bucket**

if (ht[i] == NULL) return NULL; **//if bucket itself empty then no such element in hash table**

else {

HashMapElemType \*curr = ht[i]; **//pointer for current slot**

while(curr && curr->val != v) curr = curr->link; **//searching for slot according to value v**

if (!curr) return NULL; **//if curr is NULL it means we didn’t find slot with value v**

else return curr; **//otherwise we found**

} }

**//function to insert new element**

template <class HashMapElemType> void

HashMap<HashMapElemType>::insert(const KeyType k, const ValType v)

{ **//we create and HashMapElem for new key and increase the mapsize**

HashMapElemType \*temp = new HashMapElemType;

temp->key = k; temp->val = hashfunction(k); temp->link = NULL; mapsize++;

unsigned int i = (temp->val)%capacity; **//’i’ will store our bucket**

if (!ht[hash]) ht[i] = temp; **//if i’s bucket is empty we assign new elem to first slot**

else { **//otherwise search for last slot and link with new making it last**

HashMapElemType \*curr = ht[i];

while (curr->link) curr = curr->link;

curr->link = temp;

}

}

**// function to remove the key**

template <class HashMapElemType> bool

HashMap<HashMapElemType>::remove(const KeyType k) {

unsigned int v = hashfunction(k); **//calculating the hash value of k**

unsigned int i = v%capacity; **//finding its bucket**

if (ht[hash]) { **//first we will search for this elem with val v**

HashMapElemType \*curr = ht[i]; **//pointer to the current slot**

HashMapElemType \*prev = NULL; **//pointer to the previous slot**

while(curr->link && curr->val != v) { prev = curr; curr = curr->link; }

if(curr->key == k) { **//if we found elem with key k**

mapsize--; **//we decrease mapsize**

HashMapElemType \*temp = curr->link; **//create temp pointer pointing to next slot**

delete curr; **//delete our slot**

if(!prev) ht[i] = temp; **//if it was first slot make next slot as first**

else prev->link = temp; **//if not, connect next slot with prev slot**

} } }

**// function to calculate and return hash value according to key**

template <class HashMapElemType>unsigned int

HashMap<HashMapElemType>::hashfunction(const KeyType k) {

unsigned int v = 0;

unsigned int p = 233;

unsigned int f = 1;

int n = k.size();

for(int i=0; i<n; i++) { v = v\*f + k[i]; f \*= p; }

return v; }

**// function to print our all the elements in our hash table**

template <class HashMapElemType>void

HashMap<HashMapElemType>::print() {

for(int i=0; i<capacity; i++)

if(ht[i]) std::cout<<ht[i]->key<<":"<<ht[i]->val<<" "<<i<<"\n"; }

**Spell Check**

**// function to search for the inputted word, if word is not our dictionary then output suggested words**

void spellcheck(std::string s) { std::cout << "> " << s;

**//if word is in our dictionary we just print out following**

if(HTable.find(s)) cout<<" is in the dictionary\n"; else {

**//if not, print out following, and we will try to find similar words**

cout<<" is NOT in the dictionary\n> "<<s<<" : ";

int n = s.length(); bool flag = false; **//flag is to know did we find similar words or not**

for(int i=0; i<n; i++) { **//we every letter of the word one by one starting from the left**

char t = s[i]; **//saving original initial letter**

for(char c='a'; c<='z'; c++) { **//going through every letter in alphabetical order**

s[i] = c; **//changing current position’s letter**

if(HTable.find(s)) { **//if current word exists**

if (flag) cout<<", "; **//we will put ‘,’ if before we found a word**

flag = true; cout<<s; } **//makeflag = true, print out the word**

} s[i] = t; } **//putting again original letter**

if(!flag) std::cout<<"no suggestion"; cout<<"\n"; } } **//if flag is false we couldn’t find any word**